Augmenting Metamodels with Seamless Support for Planning, Tracking, and Slicing Model Evolution Timelines

Michael Nieke, Adrian Hoff

TU Braunschweig, Germany

Christoph Seidl

ITU Copenhagen, Denmark

Ina Schaefer

TU Braunschweig, Germany

Abstract

In model-based software engineering, models are central artifacts for management, design and implementation. To meet new requirements, engineers need to plan and perform model evolution. So far, model evolution histories are captured using version control systems, e.g., Git. However, these systems are unsuitable for planning model evolution as they do not have a notion of future changes. Furthermore, formally assigning responsibilities to engineers for performing evolution of model parts is achieved by using additional tools for access control. To remedy these shortcomings, we provide a method to generate evolution-aware modeling notations by augmenting existing metamodels with concepts for capturing previous performed and planned evolution as first-class entity. To provide a clear overview, we automatically generate a Gantt-style viewer for augmented models and capabilities to slice models for certain time periods. Our method enables engineers to seamlessly plan future model evolution while actively developing the current model state using a centralized access point for evolution. With the generated Gantt-style viewers and the slicing functionality, we enable engineers to inspect relevant model evolution while reducing model size and hiding unnecessary complexity. In our evaluation, we provide an implementation of our method in the tool *TemporalRegulator3000*. We show applicability for real-world metamodels and capture the entire evolution timeline of corresponding models.

Keywords: Model Based Software Engineering, Metamodel, Model, Evolution, Planning, History, Timeline

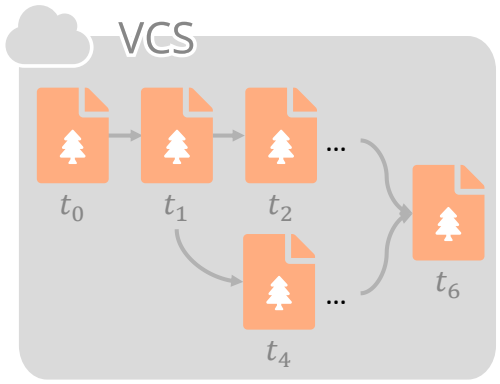
1. Introduction

In model-based software engineering, models play a pivotal role throughout the lifecycle of a project, e.g., for management, design and implementation. *Conceptual models* capture high-level concerns of a project and may be used for planning, e.g., *feature models* [1] or *process models* [2]. *Implementation models* have an operational semantics and may be the basis for code generation, e.g., state machines. To remain relevant over time, models have to evolve to meet new requirements. For this purpose, engineers perform changes on models and commit the new models to a version control system (VCS), e.g., Git [3], which aids in storing and retrieving previous model versions. We identified two major shortcomings of this practice, which we address in this paper:

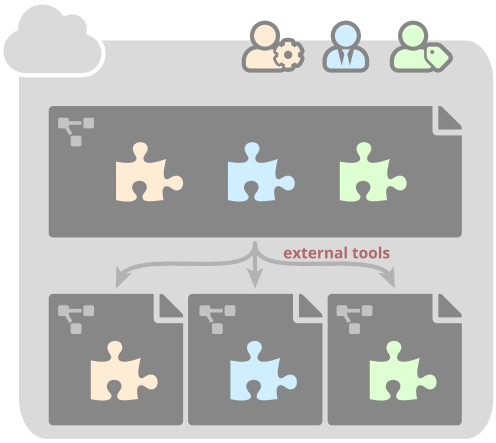
First, especially for larger systems that constitute a core strategic investment for a company, planning evolution of the respective models is crucial in defining milestones for development, monitoring evolution

Email addresses: nieke@isf.cs.tu-bs.de (Michael Nieke), adho@itu.dk (Adrian Hoff), chse@itu.dk (Christoph Seidl), i.schaefer@tu-bs.de (Ina Schaefer)

progress and performing preliminary analyses for the planned state after evolution. In particular, conceptual models or abstract parts of implementation models would lend themselves to this planning by sketching future changes without having to specify all implementation details. However, the current practice of using VCSs for evolution does not support evolution planning and can emulate it only via workarounds, e.g., by maintaining an additional branch with a planned state of a model that has to be kept in sync by repeated merging, which may require manual resolution of resulting conflicts. Figure 1a illustrates the current practice of versioning models.



(a) Versioning and branching of model evolution using version control systems.



(b) Access control of models via individual model files and external tools.

Figure 1: State of the art practices for accessing models (b) and storing their evolution (a).

Second, in larger models, different model parts are maintained by different engineers depending on their expertise and competence. To avoid inadvertent or even malicious access, changes to model parts for which an engineer does not have the responsibility must be prevented. However, with state of practice methods, models have to be decomposed into multiple files and access to these files has to be regulated using external tools as Figure 1b illustrates. This severely limits the granularity of access control as a model cannot generally be decomposed along areas of expertise, e.g., in tightly integrated interdisciplinary projects.

To address these shortcomings of current model evolution practices, in this paper, we provide a method to enable arbitrary modeling notations for integrated storing, tracking, planning, and access control of model evolution. Figure 2 briefly summarizes the goals of our contribution on an abstract level. We provide both the conceptual basis and a practical implementation to extend an existing notation's metamodel with first-class concepts for evolution support in their respective modeling notations. This procedure is fully automated and, while it yields a new metamodel, we also generate facilities that ensure seamless integration with an existing modeling infrastructure, e.g., editors. In the resulting evolution-aware model, the entire evolution timeline is stored in one sole artifact as direct connections between already performed evolution as well as planned future evolution and its later enactment. In the generated model evolution facilities, we provide a centralized access point to perform the actual changes. With this access point, we lay the foundation for element-based access control in terms of restricting changes to model parts without the need of an external access control tool. Additionally, we provide mechanisms that automatically generate Gantt-style viewers that enable engineers to inspect the entire model-evolution timeline.

For long evolution timelines, evolution-aware models have the potential to increase in size dramatically. Consequently, overview of the actual evolution is lost and past evolution steps that are not relevant for active development still exist in the model. To remedy this, we provide a method to slice evolution-aware models in chunks, e.g., to archive old model evolution steps that are irrelevant for active development or analyses. The slicing splits models into two parts, one only containing old evolution steps for archiving and one current model for active development.

In summary, we make the following contributions:

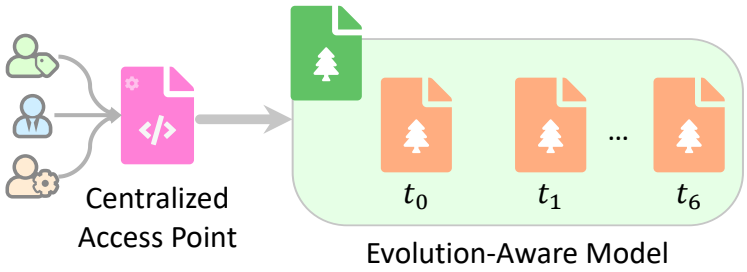


Figure 2: Evolution-aware model with centralized access point to realize access control.

- We provide a method to augment arbitrary metamodels with integrated support for model evolution.
- We automatically generate Gantt-style viewers for evolution-aware models to provide an overview on model-evolution timeline.
- We provide means to generate a centralized evolution gateway, which serves as single point of access to evolution information while hiding the complex intricacies of model evolution.
- We provide mechanisms to slice evolution-aware models at specified date to archive old evolution steps and to reduce model size.
- We enable seamlessly using evolution-aware models with existing tools by providing a set of adapters that transparently track evolution but appear similar to the original model elements regarding their interface.
- We fully automate our augmentation method, make it a generative process that is available for practical use and show its feasibility by providing an implementation within our tool *TemporalRegulator3000*.
- In our evaluation, we show that our method is applicable to real-world metamodels, that we are able to represent evolution timelines of large-scale models, and that our slicing can significantly reduce the size of evolution-aware models.

Through our augmentation mechanism for metamodels, we are able to provide evolution support for arbitrary models while striking a balance between applicability to a wide range of modeling notations, ease of use and compatibility with an existing modeling infrastructure. As result, we enable tracking evolution, active development, future evolution planning, and centralized access control in one coherent notation without the need for additional tools. This article details and greatly extends our previous work [4]. In particular, we extend this introduction with additional illustrations and extend the running example in Section 2. Moreover, we provide detailed descriptions of the mechanism to attribute model accesses with time in Subsection 4.3 and detail the core contributions in a summarizing overview in Subsection 4.5. Additionally, we provide novel concepts to slice models and provide a clear overview on a model's evolution using Gantt viewers in Section 5. We extend the implementation descriptions (Subsection 6.1) with details of the new concepts and additional information on already provided concepts. Finally, we evaluate our slicing method in Subsection 6.2. Even though we present a technical solution that deals with specifics of EMF Ecore as metamodeling notation, the concepts we provide can equally be applied to other MOF-based metamodeling notations.

2. Background and Motivating Example

As motivating example, we use the automotive Body Comfort System (BCS) [5, 6] with slightly simplified notations and evolution history. The BCS comprises functionality for the door system, the alarm system, the central locking system, and parts of the human machine interface of a car.

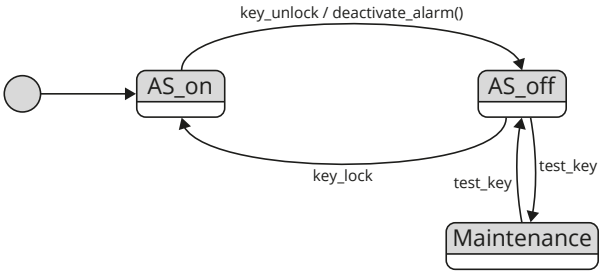


Figure 3: Exemplary state machine for the alarm system.

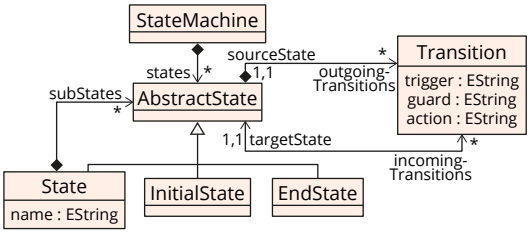


Figure 4: Metamodel of state machines.

2.1. Example Scenario

The BCS defines wide parts of its high-level functionality via state machines. A state machine models system behavior in terms of states and transitions [7]. In addition to common states, further state types exist: an *initial* state defines the start of execution; if an *end* state is reached, the system stops; *composed* states consist of sub states, i.e., they are state machines themselves. A transition may be labeled with *triggers*, *guards*, and *actions*. A trigger defines an event the transition reacts to. A guard defines a condition (usually a Boolean expression) under which a triggered transition may be activated. An action is executed when navigating to the target state.

Figure 3 shows an exemplary state machine for the alarm system of the BCS, which activates if an intrusion is detected. It consists of three states: an initial state, a state `AS_on`, which represents that the alarm system is active, and a state `AS_off`, which is active if the alarm system is deactivated. The system starts with the alarm system turned on. If the alarm system is activated and the car is opened with the key, the event `key_unlock` is fired, which activates the transition to `AS_off`. Additionally, if an alarm is active, it is deactivated if the car is opened. If the car is locked again, the event `key_lock` is fired and the alarm system is activated again. Additionally, a `Maintenance` state enables technicians to test the alarm system if a specific `test_key` event is triggered when the car is unlocked. In an early version of the system, only the high-level behavior of the alarm system is specified to define interfaces with other sub-systems, i.e., locking and unlocking of the car. The behavior for intrusion detection is not specified at this point.

2.2. Metamodel

In model-based software engineering, *models* play a pivotal role throughout the entire lifecycle of a project, e.g., for management, design, implementation, or, in the case of state machines, behavioral specification. A *metamodel* specifies a particular modeling notation in a structured way [8]. For instance, for state machines, it defines that different types of states exist, which may be connected via transitions. The Meta Object Facility (MOF) is a standard for defining metamodels in a common notation. A metamodel consists of classes, attributes and references between classes. A model is an instance of a metamodel, i.e., a concrete artifact defined using the notation specified in the metamodel. To use the modeling notation defined in a metamodel, one has to create instances of classes, which we refer to as *objects*.

Figure 4 shows an exemplary metamodel for state machines [7]. The states of a `StateMachine` are defined using the containment reference `states` to the abstract class `AbstractState`. A containment

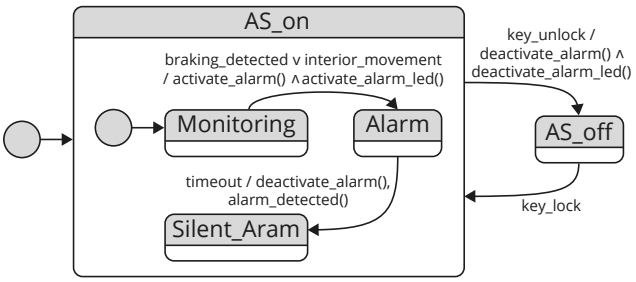


Figure 5: Planned evolution for the alarm system state machine.

reference is a special type of reference, which denotes that the containing class is owning the contained class, i.e., objects of the contained class only exist within the lifespan of objects of the containing class. A reference has a multiplicity denoting the lower and upper bounds on the number of referenced objects. The three types of states `State`, `InitialState`, and `EndState` inherit from `AbstractState`. The class `State` additionally has a `name` attribute, and if it is a composed state, its sub states are defined using the containment reference `subStates`. An attribute is similar to a reference but, instead of referencing objects of another class, it stores values of a primitive type. A `Transition` has three attributes, which define its `triggers`, `guards`, and `actions` as Strings. The source and target state of a transition are defined using respective references to `AbstractState` and the set of outgoing and incoming transitions of an `AbstractState` are defined as opposites to the `Transition`'s references.

2.3. Motivating Example

To meet new requirements or to fix bugs, system behavior needs to change and, thus, the state machines that specify this behavior need to evolve. In the motivating example's first version, the actual intrusion detection is not specified. To integrate this functionality, the state machine has to be extended. However, for large changes, this requires thorough planning and engineers should only be allowed to change parts for which they are responsible. Before implementing system behavior based on these state machines, it must be ensured that the specified behavior satisfies the requirements to avoid incorrect implementation.

Figure 5 shows the planned state machine evolution, which contains intrusion detection and alarm activation logic as sub states of the `AS_on` state. During `Monitoring`, the system reacts to the events `braking_detected` and `interior_movement` by activating the alarm and its LED. After a timeout, the system stops the alarm and logs the intrusion via the `alarm_detected()` action. The system then remains in the `Silent_Alarm` state where the LED is still active. Additionally, the behavior when unlocking the car is planned to be adapted as the LED has to be deactivated when opening the car. Finally, the `Maintenance` state is removed as testing and retrieving diagnosis data is now always possible when the car is unlocked.

Enacting planned evolution of a complex system is a large endeavor and may not be performed in an ad-hoc manner. For state machines, the implementation of abstract system behavior can generally be generated, but details have to be implemented manually. In the first evolution iteration of the example scenario, the `Monitoring` state with the intrusion detection is implemented first. If this works reliably, the other new states for activating the alarm with the timeout are implemented. Thus, the planned evolution history is successively realized with each new revision being regarded as current version of the state machine.

In contrast to planned evolution, fixing critical bugs or implementing requirements on short notice results in ad-hoc changes. In the exemplary scenario, a remote control key to lock and unlock the car is introduced shortly before the release of the system. Consequently, another trigger for the transitions between the activated and deactivated alarm system state is added. Figure 6 shows the changed state machine. It must be ensured that these changes do not lead to inconsistencies with the planned changes of Figure 5.

When using a VCS, branches for realized changes and planned evolution have to be maintained and merged manually. Conflicts between the current state and future plan become apparent only when evolution is being realized and branches are being merged. Furthermore, short-term changes have to be made compatible

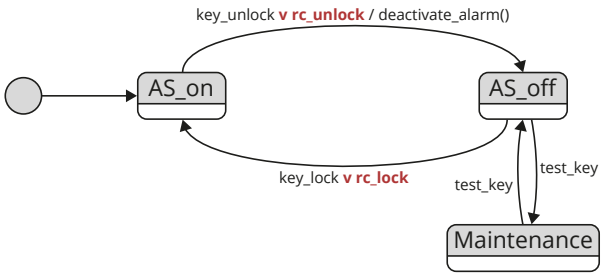


Figure 6: Intermediate development of the alarm system state machine with additional functionality for remote control keys.

with long-term plans immediately upon committing even though more sophisticated changes may have to be planned to still maintain the original plan. With an evolution-aware modeling notation that is capable of planning future evolution, consistency analyses can be performed directly when planning changes, and short-term changes may temporarily diverge from long-term plans.

In system development, multiple engineers with different competencies are involved. In the exemplary scenario, engineers who are responsible for locking and unlocking the car implement the triggers for remote control, and engineers who are responsible for the alarm system provide the sub-states for the `AS_on` state. To ensure that engineers do not interfere with each other, e.g., in safety or security critical systems, it is important to define responsibilities for different elements of a model. With existing techniques that work on file basis, it is not possible to enforce that only the alarm system engineers change the `AS_on` state. Consequently, intentional or unintentional changes to a model by engineers who do not have the competency to change this system cannot be prevented. With a notation that has a centralized access point to model changes, access control can be implemented without the need of external tools and even on granularity of individual model elements.

3. Augmenting Metamodels

As described in the motivating example, a model evolution plan is a living artifact where the planned model evolution is continuously enacted and consequently becomes the current state. With existing methods, e.g., VCSs, no support for future evolution exists, which requires to store planned model evolution as branches or snapshots. This requires to keep planned evolution and active development in sync manually.

In this article, we provide a method to precisely store model evolution as continuous timeline, which explicitly enables planning of future evolution. We augment metamodels with concepts for evolution as first-class entity. As basis, we use *temporal elements* [9] with a temporal validity $\vartheta = [\vartheta_{since}, \vartheta_{until})$ - a right-open interval that defines a time span in which the element is temporally valid, i.e., the time span from when it is first introduced to when it is decommissioned. Planning future evolution can be achieved by setting the temporal validity to dates beyond the present. To enact planned evolution, the introduction date of single elements can successively be set to the current point in time. As already performed and planned evolution are stored within the same model, the connection to other evolution steps is captured directly. Introducing intermediate changes, such as short-term changes, maintains this relation. This enables to reason about the entire model evolution timeline (i.e., past, present, and future) which could be used, e.g., to ensure consistency of short-term changes with planned evolution. In this paper, we chose real time as time unit for temporal validities, but in general, the augmentation is applicable to any time units that have a total order, e.g., version numbers or milestones.

To augment an arbitrary metamodel with evolution concepts, each element of the metamodel whose instances may be subject to change is extended by a temporal element. This includes classes, attributes and also references between classes. For this purpose, in the following, we define a set of generic transformation rules to create an evolution-aware metamodel. To ensure practical applicability, we provide transformation rules for all elements of an EMF Ecore metamodel, but our concepts are applicable to other MOF metamodel languages [8].

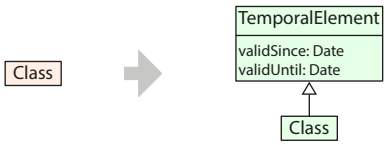


Figure 7: Transformation rule for augmenting metamodels with class evolution.

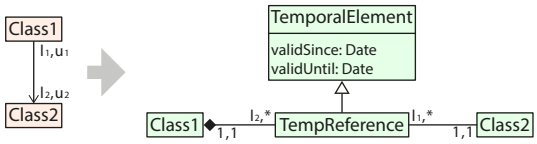


Figure 8: Transformation rule for augmenting metamodels with unordered class-reference evolution.

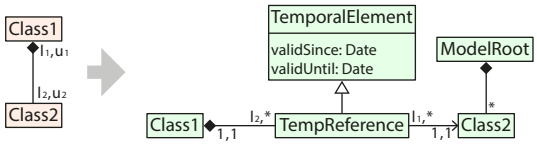


Figure 9: Transformation rule for augmenting metamodels with unordered class-containment reference evolution.

3.1. Augmentation of Classes

To plan the introduction or removal of objects (i.e., instances of classes), it is necessary to be able to define the point in time of their creation or decommission. In the motivating example, the state `Monitoring` is planned to be created (cf. Figure 5). The respective points in time can be modeled directly using the temporal validity of a temporal element. Thus, a class that should become evolution-aware needs to become a temporal element. Figure 7 shows the transformation rule for classes. After transformation, the class of the original metamodel inherits from the class `TemporalElement`. Thus, it is possible to express that, during evolution, an object is created by setting its ϑ_{since} or decommissioned by setting its ϑ_{until} respectively. For the metamodel of state machines (cf. Figure 4), this means that the evolution-aware `AbstractState` inherits from `TemporalElement` to support state evolution.

3.2. Augmentation of References

References capture the relation between objects of a model. However, these relations may change and new relations are added. In the running example, the new states `Monitoring` and `Alarm` are added along with a transition between them. To connect the states using this transition on model-level, respective values for the metamodel's reference between state and transition is required.

A reference in a metamodel is not a separate type and, thus, it is not directly possible to model it as temporal element through inheritance. As remedy, conceptually, we create an association class to capture evolution information. Practically, we create a class that wraps the original reference, but becomes evolution-aware by inheriting from `TemporalElement`. Figure 8 shows the transformation rule for augmenting a metamodel with reference evolution.

To capture the evolution of this reference, the newly created association class `TempReference` inherits from `TemporalElement`. As, over the course of time, several references of the same objects may have been added or removed, the upper bounds u_1 and u_2 of the original reference may have to be relaxed, e.g., if the upper bound of a reference's end was constrained to a specific number. Consequently, an object of type `Class1` (cf. Figure 8) may contain an arbitrary number of `TempReference` objects and a `Class2` object may be referenced by an arbitrary number of `TempReference` objects.

The illustrated transformation rule assumes that both `Class1` and `Class2` are contained by other classes. However, it may also be the case that classes are connected using containment references. Figure 9 shows the transformation rule for contained classes. As each object can only be contained by exactly one other

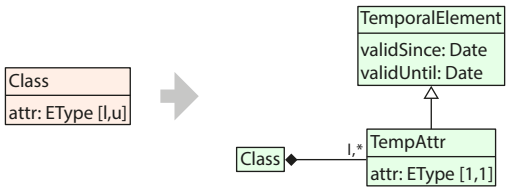


Figure 10: Transformation rule for augmenting metamodels with attribute evolution.

model element, it is not possible to define a containment reference from `TempReference` to `Class2` as objects of `Class2` may be referenced by multiple `TempReference` with different temporal validities. As remedy, a containment reference from the metamodel's root to `Class2` is defined to store all respective objects.

In the metamodel of the motivating example, references between `Transition` and `AbstractState` specify the source and target state of a transition. Thus, to define an evolution-aware metamodel that enables capturing the evolution of the source state of a transition, a new class `TempSourceState` is introduced that inherits from `TemporalElement`. Additionally, a containment reference between `AbstractState` and `TempSourceState` as well as a reference between `TempSourceState` and `Transition` are added. The original reference between `AbstractState` and `Transition` is removed.

Performing and planning reference evolution is possible by adding a new object of the `TempReference` class with respective dates as temporal validity. For instance, to change the source state of a transition at a *date*, a new `TempSourceState` object is created and its temporal validity is set to $\vartheta = [date, \infty)$. The new object holds a reference to the respective transition and a reference to the new source state. If the transition had a source state before, the temporal validity of the respective `TempSourceState` is set to $\vartheta = [\vartheta_{since}, date)$. To retrieve the source state of a transition for a particular *date*, all references to `TempSourceState` objects are retrieved and they are filtered by removing those objects that are not valid at that respective date, i.e., only keep objects for which $date \in \vartheta$ holds.

3.3. Augmentation of Attributes

In the running example, the trigger for the transition from `AS_on` to `AS_off`, which constitutes an attribute value, is modified as part of evolution (cf. Figure 6). Augmenting the metamodel with attribute value evolution works analogously to references.

Figure 10 shows the transformation rule for capturing the evolution of values of attribute `attr`. In contrast to the transformation rule for references, the new association class `TempAttr` does not have a reference to a second class but has an own attribute. The type of the new attribute matches the type of the original metamodel attribute. Each `TempAttr` object represents an (evolved) attribute value. Consequently, the multiplicity of that new attribute is 1. The multiplicity of the original attribute is captured by the multiplicity of the reference to the association class. The lower bound is equivalent to the original lower bound, whereas the upper bound is relaxed again as over the course of time, the attribute value may change several times and, thus, more values need to be captured.

In the exemplary metamodel for state machines, triggers, guards, and actions of transitions are modeled as attributes of the class `Transition`. To model the evolution of triggers, a new association class `TempTrigger` is created in the evolution-aware metamodel. To capture the evolution mentioned above, a new object of the class `TempTrigger` is created and added in the respective reference of the transition. By relaxing the multiplicity, it is possible to store the trigger before evolution and after evolution in one model. Querying and performing evolution works analogously as for the temporal references.

3.4. Augmentation of Ordered Attributes and References

In the default case, values of multi-valued references and attributes are perceived as sets without a specific order. However, if the order of values is relevant, evolution-aware models need to capture and preserve that order over the course of evolution. To this end, we provide extended transformation rules to make *ordered* attributes and references evolution-aware. For each point in time, we need to have *one*

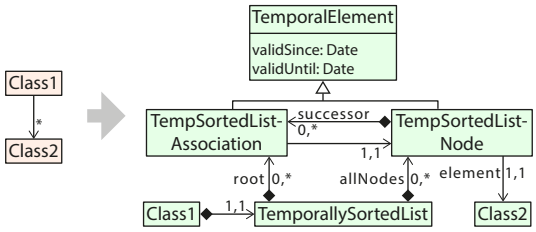


Figure 11: Transformation rule for augmenting metamodels with ordered object-reference evolution.

particular order of the list. We realize this by using linked lists. The list has to be linked for each point in time but the links may change during evolution. Thus, the list has a set of root elements and each element has a set of successor elements. However, for each single point in time, only one temporally valid root per list and one temporally valid successor element per list node is possible.

Figure 11 shows the transformation rule for ordered reference evolution. For multi-valued attributes, this works analogously. The elements of a `TemporallySortedList` are represented by `TempSortedListNode`s and the links are represented by `TempSortedListAssociation`s. As elements may be added to or removed from such a list, the `TempSortedListNode` inherits from `TemporalElement`. When adding or removing elements, new links have to be established and, thus, the `TempSortedListAssociation` inherits from `TemporalElement` as well. As the list order is subject to evolution as well, elements may have different successors at different points in time. Thus, each `TempSortedListNode` has a set of successors represented by the reference to `TempSortedListAssociation`. The same applies to the root element of the list. Note that we realized the `TemporallySortedList` using generic types, i.e., the type for a list's entries is determined by setting the generic type.

3.5. Non-Augmented Elements

For different reasons, some elements of the original metamodel do not need specific augmentation. Attributes and references may be marked with additional properties, some of which exclude augmentation: In EMF Ecore, the value of non-changeable elements cannot be altered, that of volatile and transient elements is not saved, and that of derived elements is calculated only on demand but has no own identity. In consequence, the respective values are not subject to evolution in a way that necessitates storing an evolution history. Furthermore, operations may conceptually be subject to evolution. However, we provide an engineering solution that makes it superfluous to augment individual operations, which we explain on in Subsection 4.2.

4. Automatic Generation of Access Layers

In Section 3, we introduced transformation rules to augment a metamodel for storing evolution information. However, for a practical application of our method, we have identified three problems that need to be addressed: First, if transformation rules must be applied manually, the procedure to derive an evolution-aware metamodel would be very tedious. Second, while it is technically possible to alter and query evolution information directly within evolution-aware models (e.g., for analyses or tools), accessing this information is cumbersome due to the structure of the augmented metamodel. Third, as our procedure creates a new metamodel for the evolution-aware notation, even though essential, compatibility with an existing tool landscape of the original notation is threatened.

We provide three measures to address these problems: First, we automate the application of the aforementioned transformation rules as part of a generative procedure to augment arbitrary metamodels. Second, we demonstrate how to automatically create a centralized access point for evolution, which hides intricacies of evolution-aware models both for performing evolution and querying evolution information. Third, we devise a method to automatically generate an adapter infrastructure that preserves compatibility with the original metamodel and automatically tracks performed changes by evolution information. With these contributions, we lower the barrier for both adoption and usage of evolution-aware models. Figure 12 shows an overview of the generated structure, which we use in the following to elaborate on the three measures.

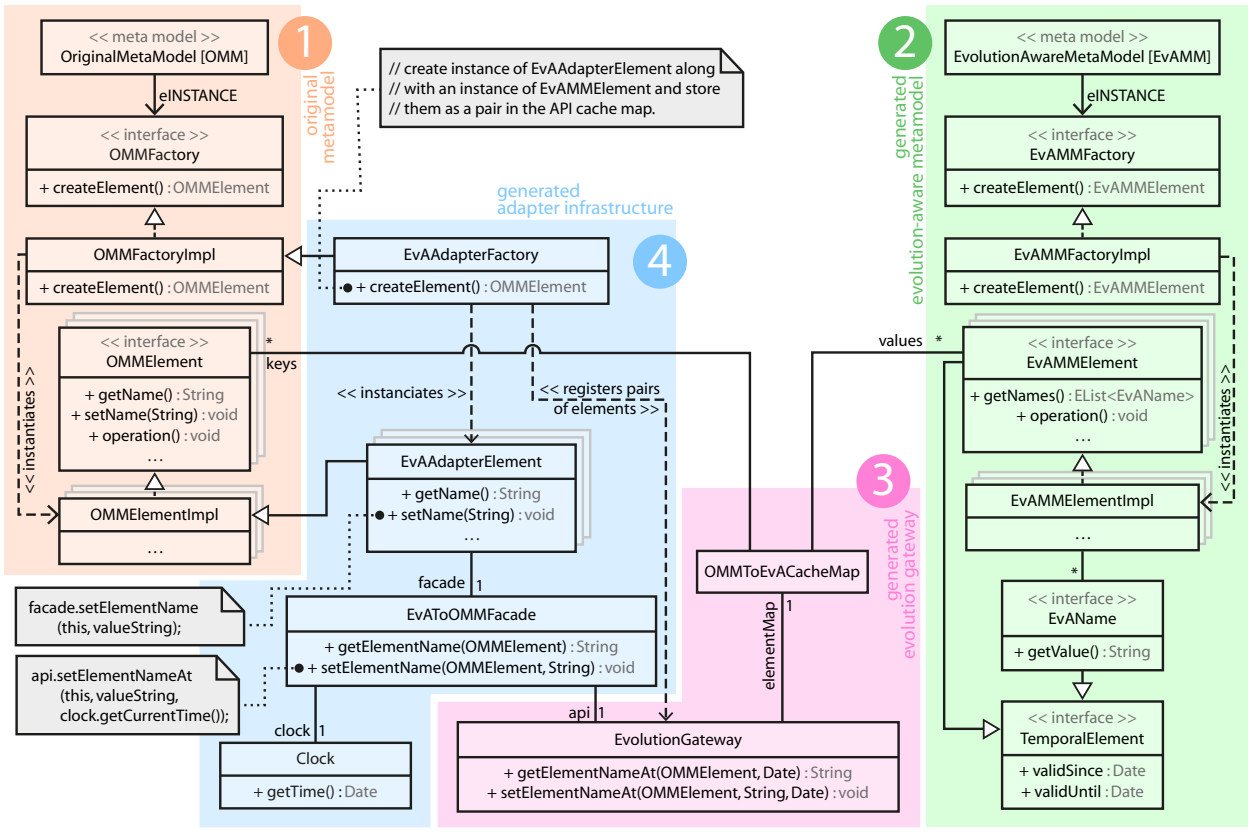


Figure 12: Generated model elements from original ① and respective evolution-aware metamodel ②. Exemplary for one class and one attribute. Evolution gateway for centralized access to evolution-aware models ③. Adapter infrastructure for transparent access using original metamodel interfaces ④.

4.1. Generating Evolution-Aware Metamodels

In Section 3, we formulated the transformation rules in a way that makes them suitable for automated application. In consequence, augmenting a metamodel in its entirety is a matter of repeatedly applying the transformation rules suitable for each respective element.

However, it is not necessarily the case that all model elements defined in a metamodel are subject to evolution. Naturally, both the procedure of augmenting a metamodel and the creation/maintenance of evolution-aware models entail a cost in the form of increased runtime and memory usage. To reduce this cost for model elements that will not be subject to evolution, we enable tailoring the generation procedure by deselecting elements of the provided metamodel. For each of the selected elements, the respective transformation rules are applied automatically to augment them. For each of the deselected elements, the original, non evolution-aware, elements are used in the augmented metamodel. The result of our automated generative procedure is an evolution-aware metamodel which enables to track, plan and analyze arbitrary evolution integrated with its respective models. Figure 12 shows the generated structure of an exemplary metamodel ① with one class (`OMMElement`) and one attribute (`name`). The right side shows the generated code of the evolution-aware metamodel after applying our transformation rules ②.

4.2. Seamless Usage of Evolution-Aware Models

Planning and performing evolution can be done by accessing the evolution-aware models directly. However, this is cumbersome due to the structures of the evolution-aware metamodel. For instance, to access the name of an `EvAMMElement` of Figure 12 at a particular date, users have to retrieve the set of all related evolution-aware `EvANames` and iterate over it to determine the name that is temporally valid at that date

(cf Section 3). This procedure is complicated even for a small example but becomes even more problematic when considering that users may be familiar with the structure of the original metamodel but not the augmented metamodel. Additionally, as our augmentation procedure yields a new metamodel, compatibility problems may arise with the infrastructure of the original metamodel, e.g., editors. Consequently, existing tools could not be used as-is to plan and perform model evolution. This results in high adoption effort and, even worse, adoption may not be possible if third-party tools are used.

To remedy these problems and to increase acceptance of evolution-aware models, we have devised a procedure that can create compatibility between the augmented and the original metamodel by automatically generating a suitable *adapter infrastructure*. To further ensure that this compatibility allows seamlessly using evolution-aware models instead of the original models even in existing implementations, it is paramount that interfaces of the original model classes are maintained. Figure 12 ③ shows an exemplary adapter infrastructure we generate. For each class of the original metamodel (`OMMElementImpl`), we generate an adapter class (`EvAAdapterElement`) that serves as proxy for accessing the data stored in the evolution-aware model. This class inherits from the original metamodel class but overrides each method for reading or writing data, i.e., getter and setter methods. Moreover, operations/methods of the original metamodel's classes are adopted by the respected adapter classes. The effect/result of an operation is based on attribute or reference values and as it now uses the respective values of the adapter element, operations always use the correct values.

For planning and performing evolution, it is necessary to specify a date for which changes are to be performed. As this general procedure is similar for all evolution-aware classes of a metamodel, we introduce a facade [10] `EvAToOMMFacade` that hides these details from the adapters [10]. The facade provides mirrored methods for each method defined in the adapter classes, i.e., setters and getters for each attribute or relation. In the adapter classes, the method calls are delegated to this facade. The facade retrieves the date for applying the changes from a central `Clock`. As default, we provide a clock that uses the system's current time. We provide an advanced override mechanism using the generation gap pattern [11], which enables supplying custom clocks, e.g., a date picker. In Subsection 4.3, we discuss this clock mechanism and provide more details. The actual execution of changes is delegated to the `EvolutionGateway`, which is introduced in the next section.

When creating a model instance of the original metamodel, e.g., using an editor, we need to ensure that instead of creating original model classes, we create adapter classes. To this end, we make use of the generation gap pattern [11] and override the existing factory (`OMMFactoryImpl`), which is responsible for creating entities, with a factory (`EvAAdapterFactory`) that creates adapter entities instead. Thus, with the adapters and the overridden factory, users can transparently use the evolution-aware model without changing anything in the implementation. As a result, changes performed to the model are tracked automatically as evolution history. Additionally, providing support for planning evolution operations is simple as only the clock has to be overridden, e.g., by using a graphical element to pick a time.

4.3. Model Access with the Clock Mechanism

Seamless usage of the augmented models by existing tools is only possible by accessing and writing for a specific time. We provide default base functionality for a clock to set a time, which does not require engineers' interaction. Moreover, we provide an extension mechanism to implement custom clocks. This can be used if a clock mechanism should be integrated into a custom user interface or directly into the editor. In the following, we detail the different levels of the clock mechanisms.

The basic clock functionality is provided by the core functionality that we generate when augmenting a metamodel. This is the default clock and uses the current system's time. If no other clock is implemented, each `EvAToOMMFacade` uses this clock's time accessing a model. However, this clock mechanism is very basic and users are limited to the current system's time. Thus, this only enables tracking of evolution.

More sophisticated use cases for augmented models encompass analyses of past model history and planning of future model evolution. However, this is not possible using the default clock. To overcome this limitation, we provide an extension mechanism to implement custom clocks and to notify other applications of clock changes. For instance, a respective clock can be implemented in a separate user interface or integrated in the model editor. We implemented the extension mechanism by providing two extension types following the extension object pattern [12]. We enable to set the clock's time by providing the `ClockExtension`

type. Extensions implementing this type must implement a method `public Date getTime()`. Whenever an augmented model is accessed using the `EvAToOMMFacade`, all extensions that registered as `ClockExtension` are queried for a time. The used time is the first time returned by any extension and overrides the time provided by the default clock. However, if no extension returns a time, the default clock's time is used. This can be the case, e.g., if no time has been selected in a custom user interface.

Retrieving the clock's date is important for all applications accessing the augmented model. With the `ClockListener` extension type, we implement the observer pattern to notify interested applications upon time change of the clock. Respective extensions must implement the method `public void handeTimeChange(Date newTime)`. Whenever the clock's time changes, all `ClockListeners` are notified. This can be used, e.g., to refresh the view of an editor.

4.4. Accessing Evolution-Aware Models

While the facade of the adapter infrastructure (cf. Subsection 4.2 and Figure 12, ④) enriches the method calls of the adapters with the time of the clock, the actual execution of the evolution operations still needs to be performed. To this end, we provide a method to generate a layer that connects the adapter infrastructure that is only aware of the original metamodel with the data structure of the evolution-aware metamodel. We denote this connecting layer `EvolutionGateway` (see Figure 12, ③). The `EvolutionGateway` is implemented using the facade pattern [10] and provides methods to access evolution information and perform evolution operations on each model element. To this end, a getter and a setter method for each attribute or reference is defined, and the respective methods are performed for a given point in time. As the adapter infrastructure is only aware of the original metamodel, the parameters in the methods are elements from the original metamodel. For instance, in Figure 12, the illustrated method `setElementNameAt` sets the name for the given element at the given date. The element provided to that method has the type `OMMElement`, i.e., an element matching the original metamodel. In our case, this is an object of type `EvAAdapterElement`.

Translating between Adapters and Evolution-Aware Models. Conceptually, the view for users of generated structures consists of elements resembling instances of the original metamodel (see Figure 12, ①). Technically, the view of the system utilizes instances of the augmented metamodel (see Figure 12, ②). Hence, operations of `EvolutionGateway` are called via methods that utilize types of the original metamodel for their parameters but are executed on elements of the evolution-aware model. To translate between these two representations, the `EvolutionGateway` holds and maintains a map (`OMMToEvACacheMap`) that relates adapter entities matching the original metamodel to entities of the evolution-aware model, i.e., the entities that store the actual evolution information. For each method call, the `EvolutionGateway` initially retrieves the evolution-aware model element `EvAMMElement` that is mapped to the original model element `OMMElement`.

Retrieving Evolution Information. For retrieving evolution information, the `EvolutionGateway` provides getter methods that retrieve the values that are temporally valid at the given time. If the respective attribute or relation in the original metamodel was a single value, i.e., the upper bound is 1, the `EvolutionGateway` also returns a single value. Similarly, for multi-valued attributes and references, the `EvolutionGateway` returns a collection holding elements of the respective type. In both cases, the method is performed in three steps. First, the evolution-aware model element that is mapped in the `OMMToEvACacheMap` to the given original model element is retrieved. Second, the set of all value elements is queried from the evolution-aware model element, which comprises all value elements that have ever been temporally valid or that will ever be temporally valid. Third, the set of value elements is filtered based on the elements' temporal validity regarding the given time. For instance, in Figure 12, the method `getElementNameAt` retrieves the `EvAMMElement` that is mapped to the given `OMMElement`. Then, all `EvANames` related to the `EvAMMElement` are retrieved. Finally, it is checked which of the `EvANames` is temporally valid at the given time and the result of the valid `EvAName`'s `getValue` is returned.

Performing Changes as Evolution. Planning and performing evolution operations can be achieved using the setter methods of the `EvolutionGateway`. This is a four-step process:

1. The evolution-aware model element is retrieved.

2. The value element that is valid at the given time t_0 is retrieved and the end of its temporal validity is set to the given time, i.e., $\vartheta_{until} = t_0$.

3. A new value element with the given value is created, its temporal validity is set to begin at t_0 and to end at the date at which the element of the second step originally ended.

4. The new value element is added to the set of all value elements.

For instance, in the method `setElementNameAt`, the currently valid `EvAName` of the `EvAMMElementImpl` is retrieved. The end of this name's temporal validity is set to t_0. Then, a new `EvAName` is created and its value is set to the given `String` parameter. Additionally, the beginning of the temporal validity of the new `EvAName` is set to t_0. Finally, the new `EvAName` is added to the set of names of the `EvAMMElementImpl`. In Ecore, an entity is removed using a utility method which unsets all references to the respective entity but does not delete it. Due to this, it is undecidable whether an entity is temporarily removed from all references, e.g., to move it in the model, or whether it is deleted permanently. Thus, the `EvolutionGateway` provides a distinguished method to delete elements, i.e., setting the end of their temporal validity.

Changes to multi-valued attributes and relations are performed on the collections that are returned using the respective getter methods. However, the `EvolutionGateway` returns collections with elements of the original metamodel and, thus, without information on evolution. Consequently, changes to that collection would not be transferred to the evolution-aware model and inconsistencies would arise. To address this problem, the `EvolutionGateway` enables users performing evolution on these sets by providing explicit methods for modifying multi-valued attributes and references. For instance, for an augmented metamodel for state machines, adding an outgoing transition to a state uses the following method `addOutgoingTransitionsAt(State state, List<Transition> transitionsToAdd, Date date)`. The gateway also retrieves and maintains the correct order of lists using the `TemporallySortedLists` (cf. Subsection 3.4) and automatically maintains opposite references. However, to prevent aforementioned inconsistencies, all list modifications must be performed by the `EvolutionGateway`. To ensure this, the `EvolutionGateway` returns immutable collections and the adapter infrastructure wraps these collections with an observer pattern, which is used to delegate modification operations via the facade to the `EvolutionGateway`.

Access Control. With the evolution gateway, we provide a centralized access point to evolution information without the need for manually handling the complex structure of the evolution-aware metamodel. The means for access resemble those of the original metamodel, as we use similar signatures of methods. With the additional information of the date, we are able to perform evolution and retrieve information pertaining evolution. Thus, we provide an easy entry point to use evolution-aware modeling notations. Additionally, a centralized access point for performing evolution opens up new possibilities, e.g., access control for performing evolution could be integrated to allow accessing selected model elements only to specific persons. We provide the basis to realize access control as each method of the `EvolutionGateway` can be modified to check whether a user that reads or writes data is allowed to do so. This enables to define access control on the granularity of model elements. However, to realize such access control, further details, such as role management, need to be integrated.

4.5. Summarizing Overview

The contributions to augment metamodels with evolution and to provide an infrastructure for seamless integration in existing tooling are manifold. To understand how the different parts relate to each other, we give an overview in Figure 13. The starting point of our contribution is the original metamodel. Based on this, we generate the entire evolution-aware model structure. First of all, we augment the metamodel itself using the transformation rules presented in Section 3. This enables to define evolution-aware models that store multiple versions of a model in one artifact. In contrast to existing methods, we do not capture each version in its entirety but annotate the different model elements with temporal validities.

This evolution-aware model is accessible via an evolution gateway (cf. Subsection 4.4). The gateway provides a single point of access to all information regarding evolution without the need to

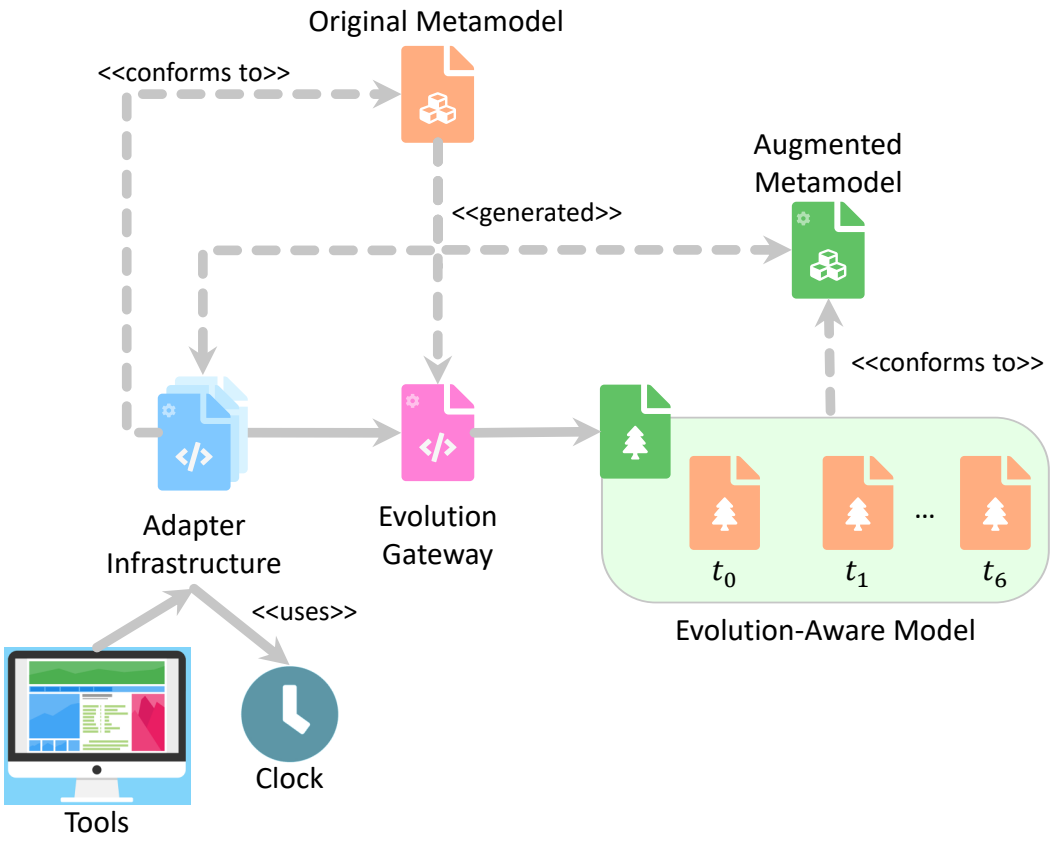


Figure 13: Overview of the core contributions and their relations.

deal with the intricacies of the evolution-aware model themselves. Moreover, this is also the point to regulate access control on model element level.

Finally, the adapter infrastructure is generated, which enables seamless access of existing tools (cf. Subsection 4.2). This is possible as the interface of the adapters conforms to the original metamodel and, thus, tools can access the adapters in the same way like original models. To translate model accesses of the adapter infrastructure to accesses for the evolution-aware model, the adapter infrasturcure uses a clock mechanism (cf. Subsection 4.3). From a clock, the adapters retrieve a date and delegate the call to the evolution gateway that finally accesses the evolution-aware model.

Figure 14 shows a chain of method calls from the adapter infrastructure to the evolution-aware model for the method `getName` on the `EvAAdapterElement`. First, the adapter delegates the call to the `EvAToOMMFacade`. The `EvAToOMMFacade` retrieves the date from the `Clock` and delegates the method call to the `EvolutionGateway` using that date. Finally, the `EvolutionGateway` accesses the evolution-aware model.

5. Optimizing Evolution timelines

Capturing the entire model evolution timeline comes with the price of large augmented models. However, keeping overview on the actual changes is challenging. During a project's life cycle, models change on a frequent basis. Core artifacts might change almost weekly [13]. Over the course of time, augmented models contain many evolution steps. Some of those evolution steps lie far in the past and, thus, become irrelevant for analyses or documentation. Consequently, those outdated evolution steps can be removed from the augmented model and stored in an archive.

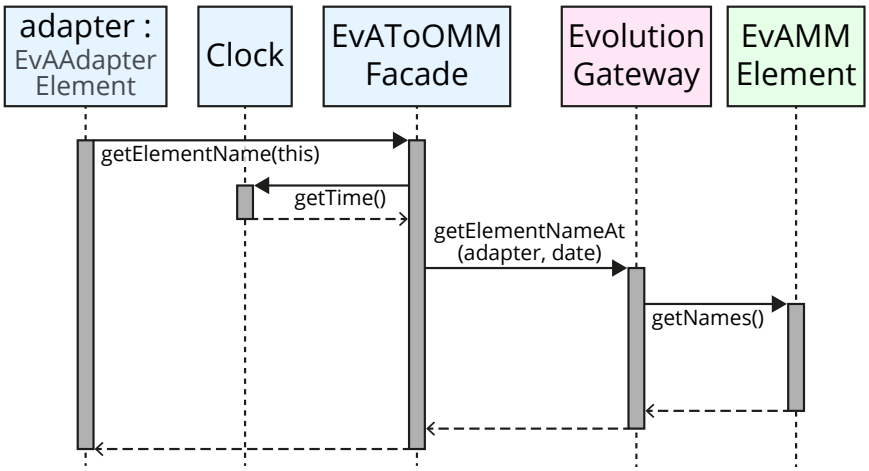


Figure 14: Sequence of method calls for adapter class with usage of facade, evolution gateway, and evolution-aware metamodel.

For large models or models with long evolution timelines, it is hard to keep overview of all changes. With the concept of the central clock (cf. Section 4), seamless access to different evolution steps of a model is possible. Nonetheless, it can be hard to understand which models elements are added, removed, or changed at an evolution step. Existing editors to create models are not aware of the evolution and, thus, do not provide means to visualize evolution and the reason for changes.

We overcome those two limitations by providing two complementary mechanisms to increase clarity of model evolution. First, we provide a Gantt-style evolution diagram viewer that provides an overview on the evolution of all augmented model's elements. Second, we introduce a slicing mechanism for augmented models that allows to separate a model's evolution timeline into multiple sub-models. In the following, we present these two mechanisms in detail and illustrate their contribution to a clear view of model evolution.

5.1. Evolution Diagrams

An augmented model provides ample information regarding evolution of all model elements. This comprises the date of element introduction or deletion as well as the evolution of attributes and relations between elements. The respective information is stored in the augmented model's temporal validity. However, assessing this information is challenging for engineers as the information is encoded in model elements. Thus, without suitable means for representation of evolution information, engineers have to laboriously retrieve this information from model elements. Additionally, even if engineers understand that an element evolved, it is unclear why it was changed or why its change is planned. We provide remedy by adding an *evolution rationale* to each temporal element during metamodel augmentation. This evolution rationale is represented by a String attribute that is contained by each temporal element. Engineers specify the intention of the performed changes in the rationales on model element level. When planning model evolution, engineers specify their thoughts on how a model should change at a future point in time. Additionally, we generate an evolution diagram viewer for each augmented augmented metamodel. This evolution diagram provides a visual overview for all augmented model elements' temporal validity and forms the basis for *understanding* an augmented model's evolution.

Figure 15 illustrates the concept of the evolution diagram using the running example state machine. In general, the evolution diagram viewer provides information similar to a Gantt chart [14] as commonly used in project management. The horizontal axis contains all dates that appear in a temporal validity of a model element, i.e., all dates at which changes were performed. To collect the time points of the temporal validities, the containment hierarchy of the evolution-aware model is iterated over. A vertical bar illustrates the current time of the clock used for evolution (cf. Subsection 4.3).

On the vertical axis, the diagram is separated into areas for each model element. A horizontal bar in each row shows the temporal validity of the respective element. The rows can be sorted via one of two options: (1) by the model's containment hierarchy or (2) by the temporal validity. Each option has certain

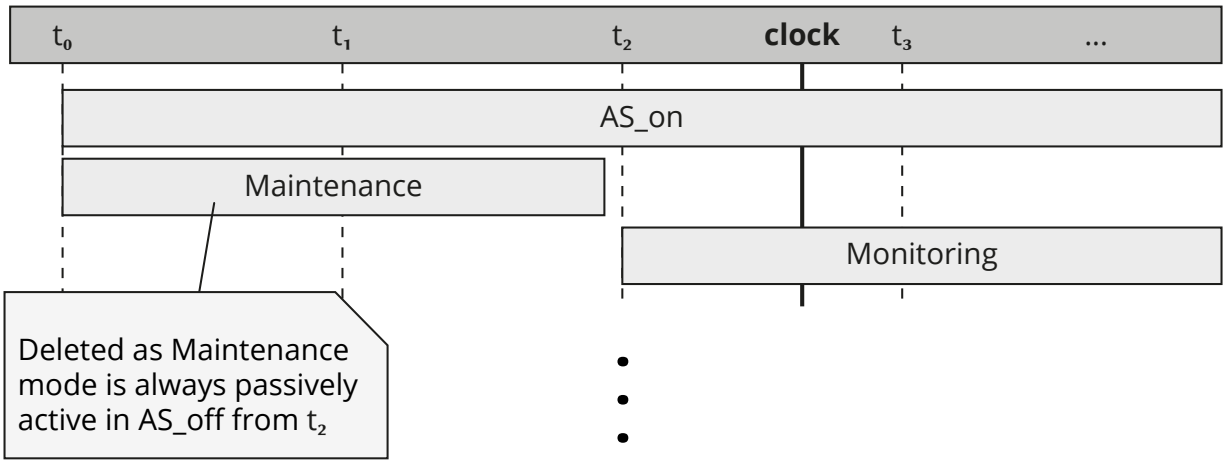


Figure 15: Illustration of the evolution diagram using the running example state machine.

benefits: With option (1), evolution of specific model elements can be determined and analyzed more easily. With option (2), the changes at each evolution step can be perceived directly.

Finally, engineers can set and view an evolution rationale for each element in this viewer. When performing evolution, engineers can specify why they added, deleted, or changed certain elements in the rationale. When analyzing a model's evolution, engineers can see the specified rationale by hovering over the respective elements. In summary, the evolution diagram provides a direct overview on a model's evolution together with its rationale. We automatically generate this viewer for each augmented metamodel. Together with the seamless integration with existing model editors using the clock mechanism, we provide means to perform and view model evolution without any additional implementation effort.

5.2. Slicing Augmented Models

Principally, augmented models lend themselves to have a good overview of evolution without the need of external tools, such as VCSs. In augmented models, evolution is not stored as snapshots, but on a timeline. Thus, it can easily be determined when an element was introduced, when it was changed, or when it was deleted. This is a great benefit compared to versioning systems. However, this method also has a drawback: As the entire evolution timeline is stored in one model, this may lead to very large models, and entities that become irrelevant are still contained in this model. For instance, old state entities of a state machine may still be relevant to analyze the model's evolution, e.g., to understand how fast a model grows. However, very old elements may become irrelevant for such analyses and can be discarded. In VCSs, this is automatically the case as the working copy of a model just contains elements that exist at that very point in time. To overcome this limitation, we provide a slicing mechanism to split an evolution-aware model with a long-term evolution history into multiple parts.

When slicing an augmented model, evolution steps are moved to the model for archiving and some steps are moved to the model for active development. As evolution steps are modeled as first-class entity in evolution-aware models, each evolution step results in certain model elements to be temporally invalid after the time of evolution step realization. Consequently, all evolution steps before the time point for slicing are moved to the archive model. Thus, if an element stops to be temporally valid before the slicing time point, it is only contained in the archive model. Analogously, elements that start to be valid after the slicing time point are only contained in the active development model. Figure 16 shows how the slicing mechanism works. In particular, we iterate over the model's containment hierarchy and perform the following steps when slicing a model m at date d, resulting in a model m_0 (i.e., model for archiving) and m_1 (i.e., model for active development):

For each entity e in m do:

- If $e_{validSince} < d$, copy e as e_0 to m_0.

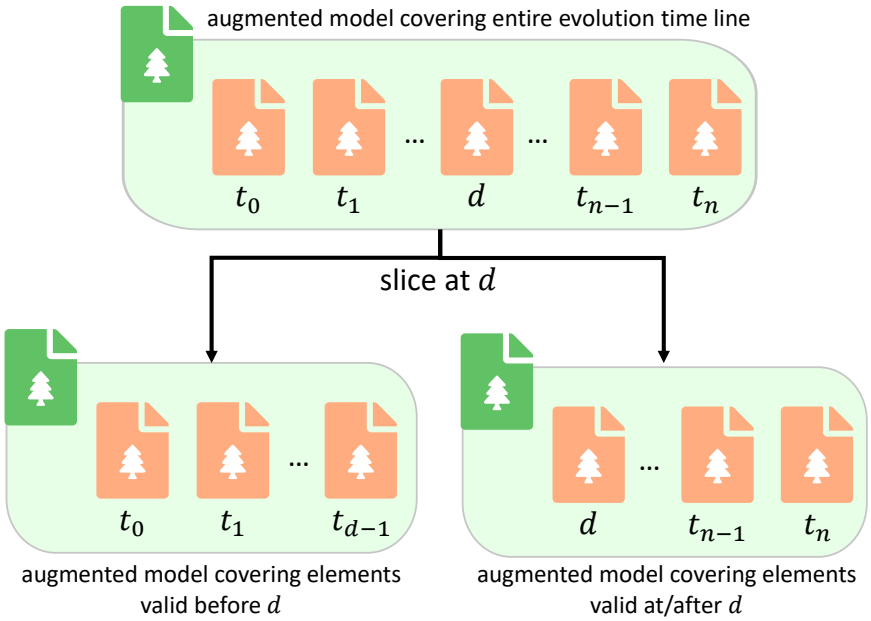


Figure 16: Illustration of slicing mechanism of an augmented model at d, resulting in two models.

  - Copy all outgoing references $r \in R_e$ of e to R_{e_0}.

- If $e_{0_{validUntil}} > d$, set $e_{0_{validUntil}} = d$.
- If $e_{validUntil} > d$, copy e as e_1 to m_1.
  - Copy all outgoing references $r \in R_e$ of e to R_{e_1}.
- If $e_{1_{validSince}} < d$, set $e_{1_{validSince}} = d$.

However, references between entities may now point to elements that are not contained anymore in the sliced model. To repair this, the following steps have to be performed for m_0 and m_1:

For each reference $r \in R_e$ of each entity e in models m_0 and m_1 with target entity t do:

- If t is not contained in the same model as e: $R_e \setminus \{r\}$

By applying this procedure, m_0 only contains elements that existed at some point before d and m_1 only contains elements that existed at some point after d. Thus, if model evolution before d became irrelevant, m_0 can be archived, e.g., in a repository. Consequently, the model size m_1 is reduced, which results in a clearer overview focusing only on relevant parts of the evolution. Additionally, model size is reduced such that analyses can be performed faster.

To illustrate how the slicing works, we use our running example (cf. Figures 3, 5, 6). We assume that the initial state machine model is modeled at date t_0, the intermediate is modeled at t_1, and the planned evolution is modeled at t_2. For brevity, we only consider the states and omit transitions etc. Table 1 shows the states that are contained in the augmented model of the running example with their temporal validities. At some point, engineers decide to archive versions that are older than t_2 as they are not relevant anymore for active analyses and development. Thus, we slice the model at t_2 as described above. The model that should be archived contains only elements that were temporally valid at some point before t_2. The model for active development only contains elements that are valid at some point at or after t_2. Tables 2a and 2b show the resulting states in the respective models before t_2 (cf. Table 2a) and after t_2 (cf. Table 2b). The tables show that the state `Maintenance` is only contained in the model before t_2 as it was deleted at t_2. Correspondingly, the states `Monitoring`, `Alarm`, and `Silent_Alarm` are only contained in the model after t_2 as they were

Table 1: States with temporal validites in augmented state machine model of running example.

	validSince	**validUntil**
AS_on	t_0	∞
AS_off	t_0	∞
Maintenance	t_0	t_2
Monitoring	t_2	∞
Alarm	t_2	∞
Silent_Alarm	t_2	∞

Table 2: States with temporal validites in sliced state machine models of running example (a) before t_2 and (b) at/after t_2.

(a)

	validSince	**validUntil**
AS_on	t_0	t_2
AS_off	t_0	t_2
Maintenance	t_0	t_2

(b)

	validSince	**validUntil**
AS_on	t_2	∞
AS_off	t_2	∞
Monitoring	t_2	∞
Alarm	t_2	∞
Silent_Alarm	t_2	∞

introduced at t_2. Additionally, the temporal validities of `AS_on` and `AS_off` are adapted to match the slicing point. The benefit of slicing only unfolds if many evolution steps with many deleted and introduced elements exist in the augmented model, which is why the model size reduction in this example is comparably low.

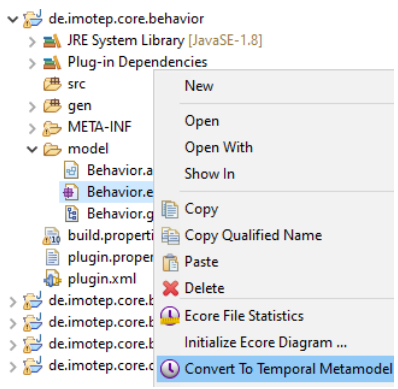
6. Evaluation

A complex software system is comprised of a multitude of different notations, e.g., UML class diagrams for modeling the static structure of a system, state machines for abstract system behavior, and Java source code for its implementation. Furthermore, safety-critical systems commonly use declarative notations to aid the certification process, e.g., Software Fault Trees (SFTs) [15]. If the system further is developed in the sense of a software product line [16], a variability model may describe its configuration options, e.g., as a feature model [17]. To show applicability of our augmentation method, we evaluate different aspects of metamodel augmentation on model-based representations of all these notations. In particular, we show that it is possible to capture and access model evolution timelines while preserving compatibility with existing tools. Additionally, we show that our method provides a basis for model-element based model access control. As a first step, we present the implementation of our augmentation method and then we present the five research questions that guide our evaluation.

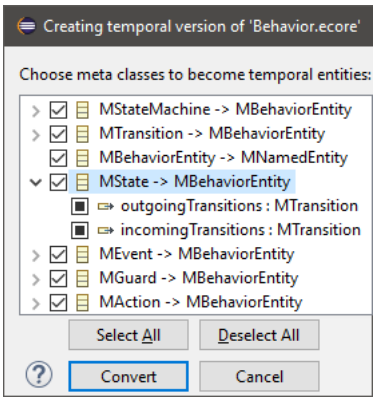
6.1. Implementation

The *TemporalRegulator3000* is the implementation of our augmentation method. It can be found in our online repository.[1] The tool generates all parts described in this paper, i.e., the augmented metamodel, the different access layers, the model-specific slicing mechanism and the evolution diagram viewer. As input, it uses

[1] `https://gitlab.com/Adomat/temporalregulator3000`



(a) Context menu to start the augmentation for a state machine metamodel.



(b) Wizard of the *TemporalRegulator3000* for a state machine metamodel.

Figure 17: Screenshots of the *TemporalRegulator3000*.

arbitrary EMF Ecore metamodels and guides users with a wizard through the generation of the augmented metamodel. As the augmentation can lead to significantly larger (meta)models (which will be investigated by **RQ4**), users may choose which elements of the original metamodel should be augmented to capture their evolution. Figure 17a shows the context menu that is used to start the augmentation process of a metamodel. After selecting the elements for augmentation, the *TemporalRegulator3000* generates the respective augmented metamodel by applying the transformation rules presented in Section 3. For instance, Figure 17b shows the wizard for an exemplary state machine metamodel together with its classes and references selected for augmentation.

By running the *TemporalRegulator3000*, a new Eclipse plug-in containing the augmented metamodel and its generated source code is created. At runtime, this code serves as substitute for the original metamodel code. Consequently, all tools using the original metamodel use the new code instead – without the need for any changes (cf. **RQ2**). For instance, for an original state machine metamodel as Figure 18 illustrates, Figure 19 shows the augmented metamodel. As the screenshots show, the metamodel increases in complexity. The detailed information is necessary to perform complex analyses, while the evolution gateway hides this complexity for common use cases.

We realized the three evolution steps of the running example (cf. Figures 3, 6, 5) using our adapter infrastructure. Figure 20 shows the evolution diagram (cf. Subsection 5.1) for the evolution of the running example's states. The dialog to define an evolution rationale can be opened by right-clicking a vertical bar of an augmented model element. As illustrated, we also set the evolution rationale of the state `Maintenance` via the dialog shown in Figure 21.

6.2. Research Questions

With our augmentation method, we provide means to store the entire evolution timeline of a model. To evaluate whether the metamodel augmentation is powerful enough to address our previously mentioned challenges, we pose five research questions.

RQ1 Is the augmented modeling notation able to track, plan, and analyze model evolution in one artifact?

RQ2 Is transparent use of the evolution-aware model possible while preserving compatibility with existing tools?

RQ3 Are we able to restrict access to model elements without external tools?

RQ4 Is the metamodel augmentation applicable to large-scale metamodels?

RQ5 How much does the slicing of model timelines reduce model complexity for real-world models?

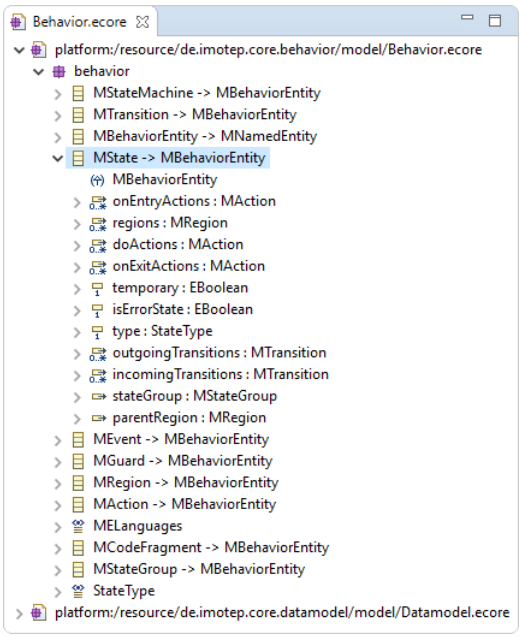


Figure 18: An Ecore metamodel for state machines.

To answer the research questions, we employ the *TemporalRegulator3000* to create augmented metamodels of the notations mentioned above. For each research question, we use a different notation to show the flexibility of our method. In the following, we describe the experiments and results. The data and tools can be found in our online repository.[2]

RQ1. We utilize the evolution history of real-world feature models [18] of a financial company.[3] The feature model's evolution history comprises ten versions and the feature model has between 557 (version 1) and 774 (version 10) features. To capture the entire evolution timeline of the feature model, we import multiple model evolution snapshots into one augmented model.

We extended the *TemporalRegulator3000* by a generic function to import multiple versions / snapshots of a model into one evolution-aware model, which only requires to implement a comparator for all metamodel elements. We successfully imported all versions into one single augmented model. By setting the `Clock`'s date to a future point in time, we imported multiple versions as planned evolution steps. We verified that we imported all versions correctly by reproducing each version from the evolution-aware model. In particular, we set the `Clock` to the date for which we imported the original version and then compared both models. In summary, we were able to track and plan model evolution in one artifact.

To show feasibility of analyses on augmented models, we investigated which feature groups changed frequently and which groups had many features added. The features in these groups seem to be the less stable ones and potentially need a lot of testing. By using the information stored in the evolution-aware model, this analysis was straight-forward to realize. For each group, we checked the association elements for the relation to its child features and their temporal validity. As we want to show only feasibility, we present numbers for the most relevant groups. We identified two groups to which 36 features were added in two evolution steps and three groups that were changed in 6 different evolution steps. Thus, evolution-aware models provide ample data that can be used for sophisticated analyses regarding evolution.

[2] In the runtime-plugins and runtime-runtime-plugins folders of `https://gitlab.com/Adomat/temporalregulator3000`

[3] `https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels/FinancialServices01`

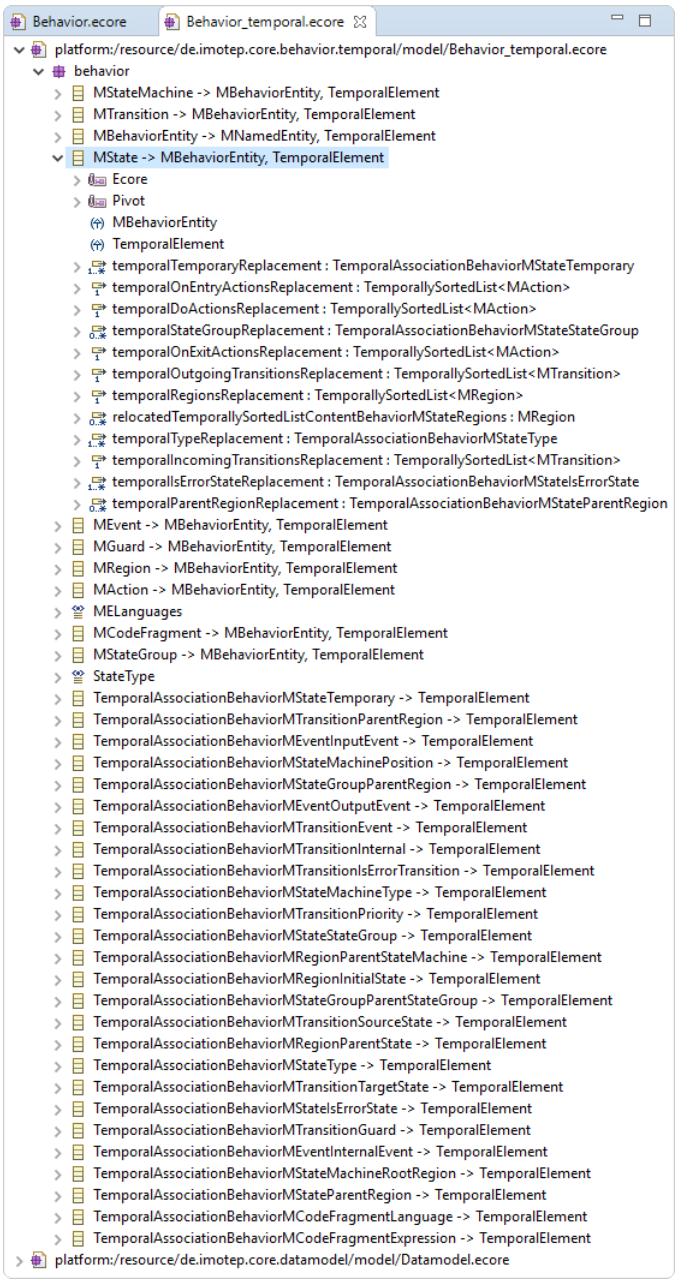


Figure 19: The augmented state machine metamodel based on the metamodel in Figure 18.

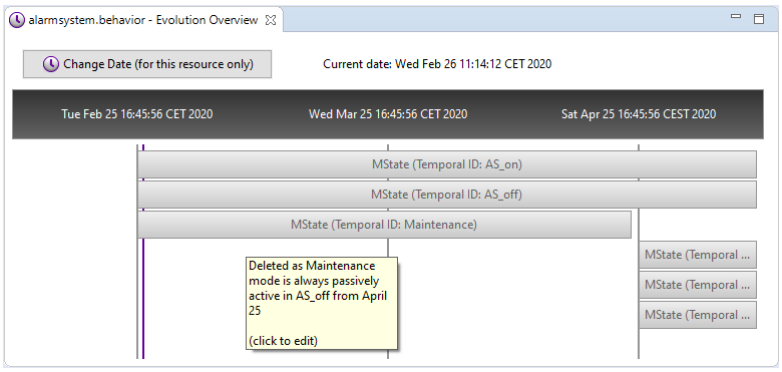


Figure 20: The evolution diagram for the running example state machine.

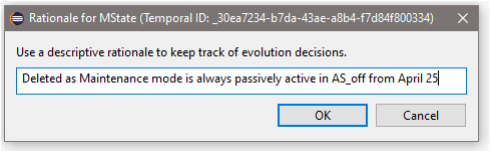


Figure 21: The dialog to change the evolution rationale of state `Maintenance`.

As the augmented metamodel introduces additional complexity, modeling a single feature version based on the augmented metamodel results in significantly increased model size. However, when considering multiple versions, the number of model elements is even reduced for the evolution-aware model when compared to individual snapshots for each version. This is due to the fact that elements which are available in more than one version are part of each model, i.e., they would be duplicated in multiple snapshots. For instance, a feature that exists in all ten versions also exists in each model snapshot. In the evolution-aware model, this feature only exists once with a temporal validity indicating for which timespan it exists. This not only reduces overall model size but also preserves the element's identity over the evolution timeline. However, by using `TemporallySortedLists` (cf. Figure 11), the number of objects increases again.

To give an estimate how evolution-aware models grow compared to the orignal models, we compared both sizes for the feature-model timeline. The sum of all objects in the feature model versions is $8,835$, the sum of all reference values is $17,640$, and the sum of attribute values is $24,837$. For the evolution-aware model, the number of objects is $14,456$ (164%), the number of reference values is $20,111$ (114%), and the number of attribute values is $18,503$ (75%). Especially the usage of `TemporallySortedLists` leads to a large increase of objects. However, we assume that the evolution-aware model will become smaller than the individual snapshots if a long evolution timeline is captured.

RQ2. Compatibility with the original metamodel notation is pivotal for acceptance of the augmented metamodels. Changes to a model performed with existing tools, such as editors, need to be stored as evolution in the augmented model. To this end, we generate the adapter infrastructure, which serves as proxy for accessing data of the evolution-aware model (cf. Subsection 4.2). To evaluate the adapters' compatibility, we generate an augmented notation and investigate whether existing tooling still works. In particular, we utilize a metamodel for Software Fault Trees (SFTs) and verify whether the existing editor still works. To set the date for the central clock, we provide a simple UI of a date picker.

The existing editor for SFTs still worked without any adaptations. Using the implemented UI interface, we set the clock's time to May 22, 2019. We created a simple SFT with two faults (`RF1`, `F1`) and one gate (`G`). Figure 22 shows the editor with the SFT together with the simple UI to set the clock's date. After that, we performed model evolution by setting the clock's date to May 23, 2019. We added a new fault (`F2`) under the already existing gate. Figure 23 shows the editor and the clock UI for this evolved model. When changing the clock back to May 22, 2019, the editor showed the model before

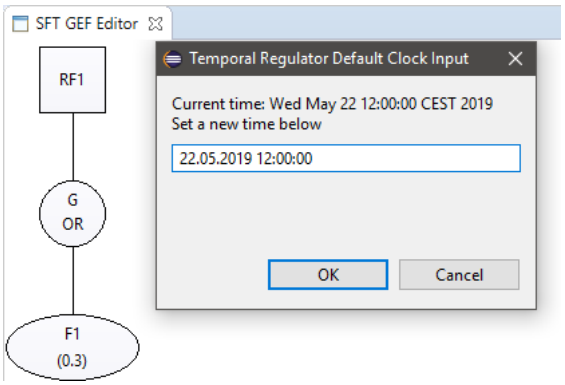


Figure 22: Screenshot of the SFT editor before evolution.

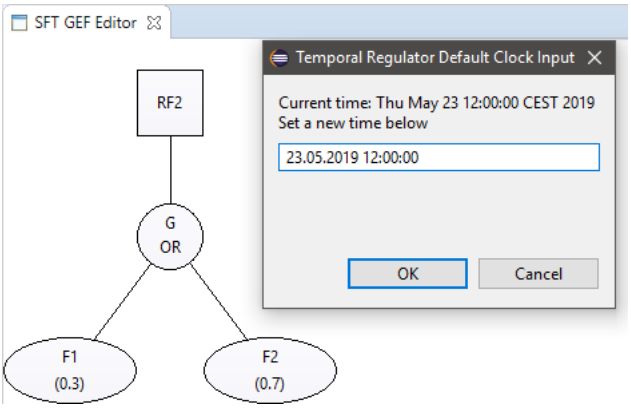


Figure 23: Screenshot of the SFT editor after evolution.

evolution again. Consequently, we are able to seamlessly integrate an augmented metamodel with existing tooling. Additionally, the evolution timeline is also stored using existing editors, which enables performing analyses regarding evolution without the need for new editors.

RQ3. Typically, access control with standard tools can only be achieved on file basis. With our augmentation method, it is possible to provide fine-grained access control on basis of individual model elements. To evaluate this, we utilize the augmented metamodel for SFTs, which we also used to answer **RQ2**. To realize such an access control, the centralized access in the `EvolutionGateway` (cf. Subsection 4.4) is modified to prohibit certain evolution operations. In particular, we disallow evolution operations modifying faults with a name containing "_critical".

Using the editor shown in Figure 22, we created a fault with name `F3_critical`. Then, we activated the access control mentioned above. Using the editor, we tried to change the name of that respective fault. In the `EvolutionGateway`, we restricted this by dropping change requests to such faults. Consequently, our actions using the editor did not result in any modifications to the SFT. Adding new faults and renaming them still worked as intended. Thus, we provide the basis for implementing fine-grained access control on element level. Using this as basis, a more sophisticated model access control system for users can be realized.

RQ4. After we evaluated the functionality of our augmentation method, we verify whether the augmentation works for large-scale real-world metamodels. To this end, we utilize two metamodels of languages used in industry. In particular, we generate augmented metamodels for the official UML2 metamodel from Ecore and for the JaMoPP metamodel for Java [19].

Table 3: Metamodel elements before and after augmentation.

		EClasses	EReferences	EAttributes
JaMoPP	Original	237	105	15
	Augmented	303	270	14
UML	Original	243	510	115
	Augmented	710	1,167	109

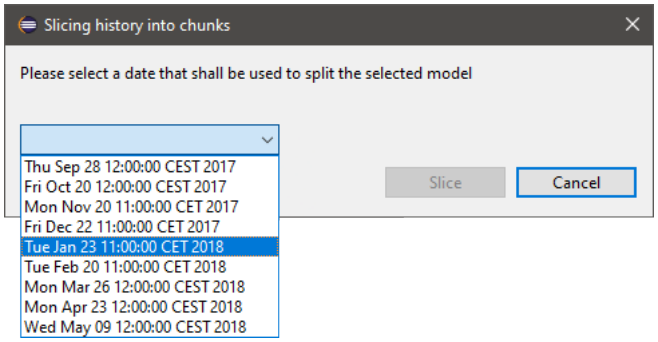


Figure 24: Dialog for slicing the evolution-aware financial services feature model.

We were able to augment the real-world metamodels for UML2 and JaMoPP. When augmenting metamodels to capture the evolution timeline, they significantly grow. Table 3 shows the size of the original metamodel compared to the augmented metamodel for UML2 and JaMoPP. The number of classes grows as augmented references and attributes are transformed into association classes (cf. Figures 8, 10). Additionally, for each new association class, new references to the source and target class of the original reference are necessary. Consequently, the number of references increases as well.

The increase in number of classes is not the same as the sum of references and attributes of the original metamodel. Moreover, the number of attributes even decreased. This is due to the fact that we use `TemporallySortedLists` with generic types to augment ordered references and attributes (cf. Figure 11). Thus, for all ordered references and attributes of the same type, we only need to add one new class `TemporallySortedList` with the respective generic type.

To give an estimate how large an augmented metamodel grows, we assume the worst case, i.e., no `TemporallySortedLists` can be used. Consequently, for each reference, a proxy class is introduced together with two references to the source and target class of the original reference. For each attribute, an association class and a reference to that class from the original class is required. As in the augmented metamodel, the attribute is moved from the original class to the association class so that the number of attributes does not change. Consequently, the increase in number of elements (classes and references) is $O(3r + 2a)$ with r being the number of references in the original metamodel (opposite references not included) and a being the number of attributes in the original metamodel. The number of classes in the original metamodel does not have any impact on the number of elements in the augmented metamodel.

RQ5. To evaluate the capabilities of our evolution-aware slicing, we apply it to the feature-model evolution data we also use for **RQ1**. The evolution-aware feature model consists of nine evolution steps. We slice the model at the fifth evolution step, i.e., removing four evolution steps to archive them. Figure 24 shows the dialog to select a time for slicing a model which can be opened via the context menu of an evolution-aware model. Table 4 shows the resulting model sizes after slicing the evolution-aware financial services feature model at the fifth evolution step. In terms of model entities and references, the size of the model for active development (i.e., the one at/after the slicing date) is reduced by 26% and the number of attributes is reduced by 27%. Thus, even with the few evolution steps in our evaluation data, our slicing method provides potential to significantly

Table 4: Model sizes after slicing evolution-aware financial services feature model at fifth evolution step.

	#Entities	#References	#Attributes
Entire time line	14,456	20,111	18,503
Before slicing date	8,443	11,719	18,036
At/after slicing date	10,664	14,863	13,432

reduce the size of evolution-aware models. The rate of reduction strongly depends on the performed evolution operations. If many entities are added but few are deleted, the size of the evolution-aware model for active development is similar to the one with the entire timeline. Vice versa, if many entities are deleted but few are added, the size of the model that should be archived is similar to the one with the entire timeline.

7. Related Work

There are multiple approaches that create evolution histories via a sequence of applied operations: Operation-based VCSs [20, 21] store evolution in version-control system by capturing the modifications that were performed instead of the state resulting from those modifications. Change-oriented programming [22] uses change objects synthesized from applied operations to represent evolutionary modifications on implementation artifacts. Engels et al. [23] and Kehrer et al. [24] capture model evolution by means of pre-defined operations or transformation scripts. The goal is to preserve model consistency between each evolution step.

A similar approach is delta modeling which enables to capture differences between artifacts such as models in separate delta artifacts [25, 26]. These delta artifacts describe how to change a specific base model and can be used to express variability but also evolution. With `DeltaEcore` a method exists to easily define delta languages for arbitrary Ecore metamodels and an infrastructure is provided to define and automatically apply deltas [27]. The main purpose of `DeltaEcore` is to enable expressing variability but with *evolution deltas*, evolution is addressed as well. Similarly, higher-order deltas enable to perform changes to existing deltas which is used to capture the evolution of deltas [28].

Unlike the approaches based on specific (delta) operations, we store model evolution internal to implementation artifacts as temporal validities, which does not limit users to a constrained set of potential operations. Additionally, our method seamlessly integrates with existing tool infrastructure and, in addition, we are able to perform planning of future evolution, which can be enacted seamlessly. Nonetheless, a combination of methods to preserve model consistency [23, 24] with our method can be sensible.

Degueule et al. [29] provide a method to enable model polymorphism in order to access a model through different DSL interfaces. Their language workbench *Melange* generates an adapter infrastructure for Ecore models that enables engineers to provide different interfaces (i.e., DSLs) to manipulate a single model. Similarly, we also enable model polymorphism using a generated adapter infrastructure but we focus on storing model evolution timelines.

Bousse et al. [30] introduce trace metamodels which enable capturing of model execution traces which are similar to model evolution. The generated domain-specific trace metamodels resemble the evolution-aware metamodels that are generated in this paper. In contrast to our method, Bousse et al. explicitly aim for efficient storage of these traces but do not consider seamless integration with tooling for existing metamodels. Concepts of this method could improve storage and handling of model evolution information.

Many researchers addressed the problem of metamodel-model co-evolution [31, 32, 33, 34, 35, 36]. Similar to the above mentioned methods, these methods define operations to modify models. The focus is to preserve compatibility of existing models with evolved metamodels. However, with these approaches, it is neither possible to capture model evolution history nor plan its future evolution.

Hermannsdörfer et al. propose a generic operation recorder for model evolution [37]. Similar to our method, the goal is to capture model evolution without the need to adapt existing tools. To this end, the operation recorder utilizes the observer mechanism of the EMF environment. Changes to EMF models are

then captured in a separate generic operation model. To retrieve a certain model version, these operations need to be applied. Thus, existing tools like analyses or viewers cannot seamlessly access specific model versions.

In other fields of research, such as differencing and merging of models versions [38, 39, 40, 41] and reverse engineering of model changes [42, 43], differences between multiple model (versions) are retroactively computed. These methods work with existing VCSs, but computing the differences is expensive and often only an approximation. Additionally, seamless integration with existing tool infrastructure is not considered and planning model evolution is out of scope.

In our previous work, we defined *Temporal Feature Models* (TFMs) that enable to store an entire timeline of a feature model's evolution in one model [9]. We introduced the concept of temporal elements that form the basis of this work. For TFMs, we manually defined the new metamodel and did not aim for seamless integration in existing modeling infrastructures.

To the best of our knowledge, no method exists which considers access control on model element level. Additionally, most of the previously mentioned methods (except for the operation recorder [37]) do not seamlessly integrate with existing tool infrastructure.

8. Conclusion and Future Work

We presented a generic method for integrated capturing of past history and planned future evolution of models by augmenting metamodels. Our augmentation rules can be applied to arbitrary metamodels, and we provide concepts for generating an infrastructure for accessing augmented models. Thus, we are able to store an entire model evolution timeline in one artifact while seamlessly integrating with existing tool infrastructure. Additionally, we provide a centralized access point to model evolution which can be used to realize a fine-grained model access control on model element level without the need of third-party tools. Furthermore, we provide a Gantt-style viewer to easily inspect the evolution of arbitrary augmented models with evolution rationales. Finally, with the slicing mechanism, we are able to split timelines of evolution-aware models for archiving.

The results we present in this paper form the basis for planning of future model evolution. However, planning long-term goals for a model still requires to insert intermediate evolution steps. As a result, inconsistencies with the already planned evolution may arise (aka *evolution paradoxes*) [44, 45]. With this work, we do not solve the problem of evolution paradoxes but provide the first step towards consistent model evolution planning. Additionally, evolution-aware constraints may be a relevant field of research, i.e., constraints that define how a model may or must evolve.

A similar problem is how to verify static model constraints, either defined by the metamodel, e.g., reference cardinalities, or by constraint languages such as OCL. As augmented models represent entire model evolution timelines and the static model constraints are to be checked for one particular model version, they cannot be directly applied to augmented models. To provide remedy, we are working on an evolution-aware model constraint language called *evOCL* that is capable to interpret evolution-aware models. To this end, *model version constraints* can be defined that specify properties an augmented model must fulfill at each point in time. This is similar to existing static model constraints, whereas *model evolution constraints* specify properties on the entire evolution timeline of an augmented model, e.g., that the value of a specific attribute may not change more frequently than once a month.

To lower the barrier to adopt augmented models, we will extend evolution diagrams with multiple functionalities. For one, the viewer will be able to group cohesive blocks of model elements, e.g., by the same temporal validities or by similar levels in the containment hierarchy. Additionally, we will extend the viewer with editing functionality, allowing to easily change the temporal validity of elements. However, this requires to prevent the inconsistencies mentioned above as introducing them inadvertently has the potential to invalidate the evolution plan. Another measure to lower the adoption barrier is to increase compatibility with existing tools that bypass the EMF infrastructure to load models and, thus, are incompatible with augmented models. To this end, we plan to adapt the slicing functionality to export an original model for a given point in time. This model can be used by collaborators who use tools that are compatible only with the original metamodel. However, we need to devise methods that enable semi-automated reintegration of modified original models in an augmented model.

Another research area is metamodel-model co-evolution which considers how to change models upon metamodel modification [46, 31, 32, 33, 34, 35, 36]. An integration of these methods with our metamodel augmentation is an interesting challenge as the changes to the original metamodel need to be lifted to the augmented metamodel. Subsequently, the metamodel-model co-evolution operations or transformation rules need to be lifted to augmented models as well.

Acknowledgments

This work was supported by the Federal Ministry of Education and Research of Germany within CrESt (01IS16043S) and by the DFG (German Research Foundation) under SPP1593: Design For Future — Managed Software Evolution.

References

[1] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela, Software diversity: state of the art and perspectives, STTT 14 (5) (2012) 477–495.
[2] J. Mendling, H. Reijers, W. van der Aalst, Seven process modeling guidelines (7pmg), Information and Software Technology 52 (2) (2010) 127 – 136.
[3] D. Spinellis, Version control systems, IEEE Software 22 (5) (2005) 108–109.
[4] M. Nieke, A. Hoff, C. Seidl, Automated metamodel augmentation for seamless model evolution tracking and planning, in: Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 68–80. doi:10.1145/3357765.3359526.
URL https://doi.org/10.1145/3357765.3359526
[5] S. Lity, R. Lachmann, M. Lochau, I. Schaefer, Delta-oriented software product line test models - the body comfort system case study, Tech. rep., TU Braunschweig (2013).
[6] S. Nahrendorf, S. Lity, I. Schaefer, Applying higher-order delta modeling for the evolution of delta-oriented software product lines, Tech. rep., TU Braunschweig (2018).
[7] D. HAREL, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[8] O. M. Group, Omg meta object facility (mof) core specification, Tech. rep., Object Management Group (2016).
[9] M. Nieke, C. Seidl, S. Schuster, Guaranteeing configuration validity in evolving software product lines, in: Proceedings of the Tenth Intl. Workshop on Variability Modelling of Software-intensive Systems, ACM, New York, NY, USA, 2016, pp. 73–80.
[10] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.
[11] J. Vlissides, J. M. Vlissides, Pattern hatching: design patterns applied, Addison-Wesley Reading, 1998.
[12] E. Gamma, The extension objects pattern, in: Proceedings of the 1996 Conference on Pattern Languages of Programs (PLoP 96), 1996.
[13] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, A. Wąsowski, Three cases of feature-based variability modeling in industry, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Model-Driven Engineering Languages and Systems, Springer International Publishing, Cham, 2014, pp. 302–319.
[14] J. M. Wilson, Gantt charts: A centenary appreciation, European Journal of Operational Research 149 (2) (2003) 430 – 437, sequencing and Scheduling. doi:https://doi.org/10.1016/S0377-2217(02)00769-5.
URL http://www.sciencedirect.com/science/article/pii/S0377221702007695
[15] N. Leveson, S. Goetsch, Safeware: System Safety and Computers, Medical Physics-New York-Institute of Physics 23 (10) (1996) 1821.
[16] K. Pohl, G. Böckle, F. J. van der Linden, Software Product Line Engineering - Foundations, Principles and Techniques, Springer Berlin/Heidelberg, 2005.
[17] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. rep., DTIC Document (1990).
[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst (1990).
[19] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, Closing the gap between modelling and java, in: M. van den Brand, D. Gašević, J. Gray (Eds.), Software Language Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 374–383.
[20] M. Koegel, M. Herrmannsdoerfer, J. Helming, Y. Li, State-based vs. operation-based change tracking, in: Proceedings of MODELS, Vol. 9, 2009.
[21] T. T. Nguyen, H. A. Nguyen, N. H. Pham, T. N. Nguyen, Operation-based, fine-grained version control model for tree-based representation, in: D. S. Rosenblum, G. Taentzer (Eds.), Fundamental Approaches to Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 74–90.
[22] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, T. D'Hondt, Change-Oriented Software Engineering, in: Proceedings of the Intl. Conference on Dynamic Languages, ACM, 2007.

[23] G. Engels, R. Heckel, J. M. Küster, L. Groenewegen, Consistency-preserving model evolution through transformations, in: J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.), UML 2002 — The Unified Modeling Language, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 212–227.
[24] T. Kehrer, U. Kelter, G. Taentzer, Consistency-preserving edit scripts in model versioning, in: 2013 28th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE), 2013, pp. 191–201.
[25] D. Clarke, M. Helvensteijn, I. Schaefer, Abstract delta modelling, Mathematical Structures in Computer Science 25 (3) (2015) 482–527. doi:10.1017/S0960129512000941.
[26] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 77–91.
[27] C. Seidl, I. Schaefer, U. Aßmann, Deltaecore-a model-based delta language generation framework, in: H.-G. Fill, D. Karagiannis, U. Reimer (Eds.), Modellierung 2014, Gesellschaft für Informatik e.V., Bonn, 2014, pp. 81–96.
[28] S. Lity, M. Kowal, I. Schaefer, Higher-order delta modeling for software product line evolution, in: Proceedings of the 7th International Workshop on Feature-Oriented Software Development, FOSD 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 39–48. doi:10.1145/3001867.3001872.
URL https://doi.org/10.1145/3001867.3001872
[29] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Safe model polymorphism for flexible modeling, Computer Languages, Systems & Structures 49 (2017) 176–195.
[30] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, Advanced and efficient execution trace management for executable domain-specific modeling languages, Software & Systems Modeling 18 (1) (2019) 385–421.
[31] G. Wachsmuth, Metamodel adaptation and model co-adaptation, in: E. Ernst (Ed.), ECOOP 2007 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 600–624.
[32] W. Kessentini, H. Sahraoui, M. Wimmer, Automated metamodel/model co-evolution using a multi-objective optimization approach, in: Proceedings of the 12th European Conference on Modelling Foundations and Applications, Springer-Verlag, Berlin, Heidelberg, 2016, pp. 138–155.
[33] K. Garcés, F. Jouault, P. Cointe, J. Bézivin, Managing model adaptation by precise detection of metamodel changes, in: R. F. Paige, A. Hartman, A. Rensink (Eds.), Model Driven Architecture - Foundations and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 34–49.
[34] M. Herrmannsdoerfer, S. Benz, E. Juergens, Cope - automating coupled evolution of metamodels and models, in: S. Drossopoulou (Ed.), ECOOP 2009 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 52–76.
[35] L. M. Rose, D. S. Kolovos, R. F. Paige, F. A. C. Polack, Model migration with epsilon flock, in: L. Tratt, M. Gogolla (Eds.), Theory and Practice of Model Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 184–198.
[36] A. Cicchetti, D. D. Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: 2008 12th Intl. IEEE Enterprise Distributed Object Computing Conference, 2008, pp. 222–231.
[37] M. Herrmannsdoerfer, M. Koegel, Towards a generic operation recorder for model evolution, in: Proceedings of the 1st Intl. Workshop on Model Comparison in Practice, ACM, New York, NY, USA, 2010, pp. 76–81.
[38] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer, An introduction to model versioning, in: Proceedings of the 12th Intl. Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 336–398.
[39] C. Brun, A. Pierantonio, Model differences in the eclipse modeling framework, UPGRADE, The European Journal for the Informatics Professional 9 (2) (2008) 29–34.
[40] E. Foundation, Emf compare project (2019).
URL http://www.eclipse.org/emf/compare
[41] G. Taentzer, C. Ermel, P. Langer, M. Wimmer, A fundamental approach to model versioning based on graph modifications: from theory to implementation, Software & Systems Modeling 13 (1) (2014) 239–272.
[42] Z. Xing, E. Stroulia, Refactoring detection based on umldiff change-facts queries, in: 13th Working Conference on Reverse Engineering, 2006, pp. 263–274.
[43] D. Wille, T. Runge, C. Seidl, S. Schulze, Extractive software product line engineering using model-based delta module generation, in: Proceedings of the Eleventh Intl. Workshop on Variability Modelling of Software-intensive Systems, ACM, New York, NY, USA, 2017, pp. 36–43.
[44] A. Hoff, M. Nieke, C. Seidl, E. H. Saether, I. M. Sandberg, C. C. Din, I. C. Yu, I. Schaefer, Consistency-preserving evolution planning on feature models, in: Proceedings of the 24th International Systems and Software Product Line Conference - Volume A, SPLC '20, Association for Computing Machinery, New York, NY, USA, 2020. doi:https://doi.org/10.1145/3382025.3414964.
[45] M. Nieke, C. Seidl, T. Thüm, Back to the future: avoiding paradoxes in feature-model evolution, in: Proceeedings of the 22nd Intl. Systems and Software Product Line Conference - Volume 2, Gothenburg, Sweden, September 10-14, 2018, 2018, pp. 48–51.
[46] F. Basciani, D. Di Ruscio, L. Iovino, A. Pierantonio, Automated chaining of model transformations with incompatible metamodels, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Model-Driven Engineering Languages and Systems, Springer International Publishing, Cham, 2014, pp. 602–618.